White-box Atomic Multicast

Alexey Gotsman
IMDEA Software Institute

Anatole Lefort
Télécom SudParis

Gregory Chockler
Royal Holloway, University of London

***Abstract*—Atomic multicast is a communication primitive that delivers messages to multiple groups of processes according to some total order, with each group receiving the projection of the total order onto messages addressed to it. To be scalable, atomic multicast needs to be genuine, meaning that only the destination processes of a message should participate in ordering it. In this paper we propose a novel genuine atomic multicast protocol that in the absence of failures takes as low as 3 message delays to deliver a message in the collision-free case, and no more than 5 message delays regardless of collisions. This improves the collision-free and worst-case latencies of both the fault-tolerant version of classical Skeen's multicast protocol (6 and 12 message delays respectively) and its recent improvement by Coelho et al. (4 and 8 message delays respectively). To achieve such low latencies, we depart from the typical way of guaranteeing fault-tolerance by replicating each group with Paxos. Instead, we weave Paxos and Skeen's protocol together into a single coherent protocol, exploiting opportunities for white-box optimisations. We experimentally demonstrate that the superior theoretical characteristics of our protocol are reflected in practical performance pay-offs.**

I. Introduction

Machine crashes are a fact of life in modern cloud services. The classical way of enabling the services to tolerate such failures is using a state-machine replication approach [35]: a service is defined by a deterministic state machine and is run on several replicas, each maintaining its own local copy of the machine. Different copies can be kept in sync using an *atomic broadcast* protocol, which delivers *application messages* to replicas in some total order and thereby ensures that they evolve in the same way. Unfortunately, it is often impossible for a single machine to store the whole service state. A solution is to partition the service across several *process groups*, each containing several replicas to guarantee fault-tolerance. In this setting, replica consistency can be maintained using *atomic multicast* [13]. This accepts application messages together with sets of groups they are relevant to and delivers messages to their destination groups according to some total order, so that each group receives the projection of the total order onto messages addressed to it (§II). Atomic multicast thus generalises atomic broadcast, since it provides the same guarantees in the case when there is a single process group.

Ideally, we want an atomic multicast protocol to be *genuine*, i.e., only the processes in the destination groups of a message should participate in ordering it [19]. This allows messages to disjoint sets of groups to be ordered in parallel, thus enabling scalability. For example, genuine atomic multicast has been used to scale fault-tolerant transaction processing systems [12, 30, 34] and log-based systems [27]. Genuine atomic multicast essentially requires constructing a total order on application messages addressed to different groups in a decentralised way. Achieving this is challenging, and classical implementations of genuine atomic multicast have suboptimal performance. In this paper we set out to improve this situation. Our main goal is to improve the *collision-free latency* of atomic multicast, which is roughly, the maximum time required to deliver a message in a failure-free synchronous run in the absence of interference by concurrently arriving messages.

The most well-known protocol for atomic multicast is folklore Skeen's protocol (described, e.g., in [19]), which handles a restricted setting where each group consists of a single reliable process (§III). In a nutshell, the protocol creates a total order on application messages by assigning them unique timestamps, computed similarly to Lamport clocks [23]. To multicast an application message, a client process sends it to all its destinations. Each destination process generates a *local timestamp* from a local logical clock and sends it to the other destination processes. When a process receives all local timestamps for a given message, it computes its final *global timestamp* as their maximum and advances its clock to be no lower than the timestamp. A process can deliver an application message once it is sure that no message will get a lower global timestamp. Hence, in the absence of collisions, Skeen's protocol requires 2 message delays to deliver an application message: one to send the message from the client process to the destinations, and the other for the destinations to exchange local timestamps.

A common approach to making Skeen's protocol fault-tolerant [17, 31] is to get every group to simulate a reliable process in Skeen's using a replication protocol, such as Paxos [24]. In this case each of the two key actions of Skeen's protocol—computing a local timestamp and advancing the clock above a global timestamp—requires a round trip from the Paxos leader of each destination group to a quorum of processes in the same group, to persist the effect of the action. The resulting protocol takes 6 message delays to deliver an application message in a failure and collision-free run—an extremely high latency, especially when multicast is used in a wide-area network.

In this paper we present a novel fault-tolerant atomic multicast protocol that lowers the collision-free latency of delivery to 3 message delays at the leaders of destination groups and 4 at all other processes (§IV). This improves on a recent optimised version of Skeen's protocol by Coelho et al. [10], which requires one extra message delay. Our protocol is also efficient in terms of its *failure-free latency*, which bounds the worst-case message delivery time in the presence

of concurrently arriving application messages. In particular, it achieves the failure-free latency of just 5 message delays as opposed to 8 message delays of [10], thus reducing the 2x latency degradation exhibited by all existing variants of Skeen's protocol in the presence of concurrency.

To achieve such low latencies, we depart from the standard designs of fault-tolerant multicast protocols, which have used consensus as a black box [10, 17, 29, 31]. Instead, we combine the ideas from Skeen's protocol with those of Paxos into a single coherent protocol. This allows us to exploit several white-box optimisations that lead to a more efficient solution.

In more detail, our protocol takes the *passive replication* approach [21, 28]: a special *leader* process in each group computes the timestamps and decides when to deliver an application message like in Skeen's protocol; the rest of the processes merely *follow* its decisions. To replicate leader actions when multicasting an application message, the protocol performs a message exchange similar to the one of Paxos, but between all leaders of the destination groups on the one hand and majorities of followers in all destination groups on the other. This message exchange replicates both of the key actions of Skeen's protocol—assigning a local timestamp and advancing the clock above the global timestamp—in a single round trip, thus minimising delivery latency. Since in our protocol the leader takes decisions about delivery unilaterally, based on its local state, every decision it takes on a message only makes sense in the context of its previous decisions on other messages. This requires care when recovering from a leader failure: recovery cannot be done for each application message independently (like in multi-Paxos [24]), but has to be done for all messages at once (like in Viewstamped Replication [28] and Zab [21]). We rigorously prove that our white-box protocol is correct (§V and [18]).

We also experimentally demonstrate that the superior theoretical characteristics of our protocol are reflected in practical performance pay-offs (§VI). Our protocol outperforms the state-of-the-art protocol by Coelho et al. [10] on latency and throughput by factors of up to 2.5x.

II. Problem Statement

We consider an asynchronous message-passing system consisting of a finite set of N processes $\mathcal{P}$, which can fail by crashing. A process is *correct* if it never crashes, and *faulty* otherwise. Processes are connected by reliable FIFO channels, i.e., messages are delivered in the FIFO order, and every message sent by a process p to another process q is guaranteed to be eventually delivered by q provided both p and q are correct.

We fix $\mathcal{G} \in 2^{\mathcal{P}}$ to be a set of *process groups* and let $|\mathcal{G}| = k$. We assume that the process groups are disjoint, i.e., $\forall g_1, g_2 \in \mathcal{G}.\, g_1 \cap g_2 = \emptyset$. Every group $g \in \mathcal{G}$ consists of $2f + 1$ processes, at most f of which can fail. We call a set of $f + 1$ processes in g a *quorum* in g. The assumption of disjoint groups is standard for practical multicast protocols [10, 17, 29]. It captures common usage scenarios in which atomic multicast is deployed for replicating a partitioned data store [12, 30, 34], and it does not prevent collocating processes that are members of different groups on the same machine.

We consider the problem of implementing atomic multicast in the above system, which allows a process to send an *application message* m from a set $\mathcal{M}$ to a set of *destination groups* $\mathsf{dest}(m) \subseteq \mathcal{G}$. We denote the events of multicasting a message m and delivering it by $\texttt{multicast}(m)$ and $\texttt{deliver}(m)$, respectively. For simplicity, we assume that all messages multicast in a single execution are unique. A message m is *partially delivered* if it is delivered by some process in each of its destination groups. A message m is *concurrent* with a message m' if m' is multicast before m is partially delivered, and m is multicast before m' is partially delivered. Two messages m and m' are *conflicting* if $\mathsf{dest}(m) \cap \mathsf{dest}(m') \neq \emptyset$.

An algorithm is a correct implementation of atomic multicast if its every run satisfies the following:

- **Validity.** If a process in a group g delivers a message m, then some process has multicast m before and $g \in \mathsf{dest}(m)$.
- **Integrity.** Every process delivers a message at most once.
- **Ordering.** There exists a total order $\prec$ on the set of all messages multicast in the run such that, if a process p delivers m, then for all messages $m' \prec m$, p delivers m' before m provided $p \in g$ for some $g \in \mathsf{dest}(m')$.
- **Termination.** For every message m, if m is either multicast by a correct process or delivered by any process, then for all groups $g \in \mathsf{dest}(m)$, m is eventually delivered by all correct members of g.

In particular, the ordering property ensures that each group receives the projection of a single total order onto messages addressed to it.

A protocol implementing atomic multicast is *genuine* [17, 19] if it satisfies the following *minimality* property in every run: if m is multicast in the run, then for every process p that participates in ordering m, the process p is either m's sender or a member of some $g \in \mathsf{dest}(m)$.

By instantiating atomic multicast with a single group comprising all processes in $\mathcal{P}$ we get *atomic broadcast* [20], which delivers messages to all processes. Since atomic broadcast is equivalent to consensus [7], it cannot be implemented in an asynchronous environment with failures [16]. To circumvent this impossibility, we assume that the system eventually becomes *failure-free*, i.e., the process failures cease to occur and message delays are upper-bounded by an a priori fixed constant δ. *Global stabilization time* (GST) [15] is the time (unknown to the algorithm) such that the onset of a failure-free period is guaranteed to occur no later than at GST in every run.

To measure time complexity of an atomic multicast implementation, we assign every event in a run a non-decreasing real-valued time such that after GST, the time elapsing between every pair of matching send and receive events of a protocol message is at most δ, and every step executed locally by a process is instantaneous. For a message m multicast in a run, and a group $g \in \mathsf{dest}(m)$, m's *delivery latency* with respect to g is the time elapsing between $\texttt{multicast}(m)$ and the earliest $\texttt{deliver}(m)$ by some process in g. An atomic

```
1  clock ← 0 ∈ ℕ;
2  Phase[] ← (λk. START) ∈
               (ℳ → {START, PROPOSED, COMMITTED});
3  LocalTS[] ∈ ℳ → (ℕ × 𝒢);
4  GlobalTS[] ∈ ℳ → (ℕ × 𝒢);
5  Delivered ← (λk. FALSE) ∈ ℳ → {FALSE, TRUE}

6  multicast(m)
7      send MULTICAST(m) to dest(m);

8  when received MULTICAST(m)
9      clock ← clock + 1;
10     LocalTS[m] ← (clock, g_0);
11     Phase[m] ← PROPOSED;
12     send PROPOSE(m, g_0, LocalTS[m]) to dest(m);

13 when received PROPOSE(m, g, Lts(g))
     for every g ∈ dest(m)
14     GlobalTS[m] ← max{Lts(g) | g ∈ dest(m)};
15     clock ← max{clock, time(GlobalTS[m])};
16     Phase[m] ← COMMITTED;
17     forall {m' | Phase[m'] = COMMITTED ∧
                    Delivered[m'] = FALSE ∧
                    ∀m''. Phase[m''] = PROPOSED ⟹
                          LocalTS[m''] > GlobalTS[m']}
         ordered by GlobalTS[m'] do
18         Delivered[m'] ← TRUE;
19         deliver(m');
```

Fig. 1. Skeen's protocol at a process $p_i \in g_0$.

multicast protocol has a *failure-free latency* of Δ if for every run there exists a time $t \geq \mathsf{GST}$ such that for every application message m multicast after t, m's delivery latency is at most Δ with respect to all groups in $\mathsf{dest}(m)$. A protocol has a *collision-free latency* of Δ if for every run, there exists a time $t \geq \mathsf{GST}$ such that for every application message m multicast after t that does not conflict with any concurrent messages multicast by correct processes, m's delivery latency is at most Δ with respect to all groups in $\mathsf{dest}(g)$. Note that our latency metrics are computed based on the first delivery of a message in every destination group, whereas metrics used in previous work use the last one [31]. Our choice more faithfully reflects the client-perceived latency in practical use cases of multicast, where the first process that delivers a message can process it and reply to the client [12, 30, 34].

III. Skeen's Protocol

We first consider an idealised setting where each group in $\mathcal{G}$ consists of a single reliable process. In this setting, genuine atomic multicast can be implemented using folklore Skeen's protocol (described, e.g., in [19]). This protocol serves as a basis for our optimised fault-tolerant protocol and, hence, we review it first. We give its pseudocode in Figure 1.

The protocol creates a total order on application messages by assigning them unique timestamps, computed similarly to Lamport clocks [23]. Timestamps are pairs (t, g) of a non-negative integer $t \in \mathbb{N}$ and a group identifier $g \in \mathcal{G}$. They are ordered lexicographically using an arbitrary total order on $\mathcal{G}$, with a special timestamp $\perp$ being the minimal timestamp. For a timestamp $ts = (t, g)$ we let $\mathsf{time}(ts) = t$.

To multicast an application message m, a process sends it in a MULTICAST message to the destination groups $\mathsf{dest}(m)$ (line 6). Each process maintains an integer clock, used to generate timestamps. When a process in a group g_0 receives MULTICAST(m) (line 8), it increments the clock and computes a *local timestamp of m at group g_0* as the pair of the resulting clock value and the group identifier g_0. This timestamp can be viewed as g_0's proposal of what the final timestamp of m should be; it is stored in a LocalTS array[1]. The process keeps track of the status of application messages being multicast in an array Phase, whose entries initially store START. When the process computes a local timestamp for m, it advances m's phase to PROPOSED. It then sends the local timestamp in a PROPOSE message to all the destinations of m (including itself, for uniformity).

A process that is a destination of m acts once it receives a PROPOSE message for m from each destination group $g \in \mathsf{dest}(m)$, which carries m's local timestamp $Lts(g)$ at g (line 13). The process computes the final *global timestamp* of m as the maximal of its local timestamps and stores it in a GlobalTS array. The process also advances the phase of m to COMMITTED and ensures that its clock is no lower than the first part of the global timestamp. Note that all destinations of m will receive the same sets of local timestamps for m and will thus compute the same global timestamp. Additionally, global timestamps are unique for each application message: if two messages got the same global timestamp (n, g), then they must have got the same local timestamp from group g; but this is impossible because a process increments its clock when issuing a local timestamp (line 9).

Having computed the global timestamp for m, the process tries to deliver one or more committed messages (line 17). A Boolean array Delivered keeps track of whether a given message has been delivered. Messages are delivered in the order of their global timestamps; hence, the process can deliver a message m' only if it has already delivered all messages addressed to it with a lower global timestamp. A subtlety is that the process does not know the global timestamps for the messages m'' that are in the PROPOSED phase. Hence, the process only delivers m' if all such messages m'' have local timestamps higher than the global timestamp of m': then their global timestamps will also be higher than that of m'. Note that this check is complete: application messages the process will receive for multicasting after delivering m' will get global timestamps higher than $\mathsf{GlobalTS}[m']$. This is because, when the process committed m', it advances its clock so that it is no lower than $\mathsf{GlobalTS}[m']$ (line 15). Thus, any application message the process receives afterwards will get

[1]To aid understanding, in this paper we capitalise the names of arrays and vectors.

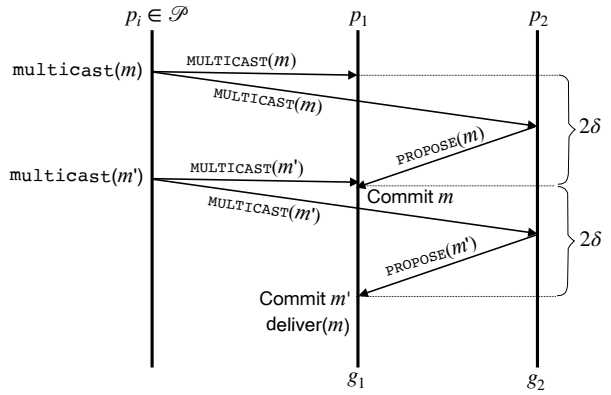

Fig. 2. Message-flow diagram illustrating the convoy effect in Skeen's protocol.

a local timestamp at g_0 higher than $\mathsf{GlobalTS}[m']$ and, thus, will also get a global timestamp higher than $\mathsf{GlobalTS}[m']$.

Theorem 1: Skeen's protocol in Figure 1 is a genuine implementation of atomic multicast among singleton groups.

Note that in Skeen's protocol a process can increase its clock at any time without violating correctness. In §IV we use this insight to construct a fast fault-tolerant version of this protocol.

Skeen's protocol has the collision-free latency of 2δ (MULTICAST, PROPOSE). However, its failure-free latency is higher because in this protocol a committed message m is blocked from delivery as long as there are any uncommitted messages with a local timestamp lower than m's global timetstamp. As a result, m's delivery latency at a process p_i may exceed the collision-free latency of 2δ in case an application message is received before the p_i's clock has been advanced past m's global timestamp—a phenomenon known as a *convoy effect* [6].

The exact amount of extra delay depends on the timing of the arrival of a conflicting message m', and can, in the worst case, be as high as 2δ. This is demonstrated by the scenario in Figure 19, where the $\texttt{MULTICAST}(m')$ message, triggered by $\texttt{multicast}(m')$ with $\mathsf{dest}(m') = \{g_1, g_2\}$, is received by p_1 immediately before m is committed at this process. Since p_1's clock is still lower than $\mathsf{GlobalTS}[m]$ at the time m' is received, this message is assigned a local timestamp less than $\mathsf{GlobalTS}[m]$. As a result, the delivery of m must now be delayed until m' commits. In the worst-case scenario of Figure 19 this takes another 2δ, because $\texttt{MULTICAST}(m')$ takes close to 0 to arrive at p_1, but exactly δ to arrive at p_2; then $\texttt{PROPOSE}(m')$ from p_2 also takes exactly δ to arrive at p_1. Thus, the failure-free latency of Skeen's protocol is in fact 4δ, i.e., double its collision-free latency.

IV. White-box Protocol

We now consider the general setting where each group consists of $2f + 1$ processes, out of which at most f can fail. A straightforward way to implement atomic multicast in this setting is to use state-machine replication to make a group simulate a reliable process in Skeen's protocol [17]; this is usually based on a consensus protocol such as Paxos [24]. Then in addition to MULTICAST and PROPOSE messages, the resulting protocol requires two round trips from the Paxos leader of a group to a quorum of processes in the same group—one to persist the local timestamp (line 10 in Figure 1) and another to persist the global timestamp and update the clock (lines 14-15). Hence, the resulting multicast protocol has the collision-free latency of 6δ; as we show in §V, its failure-free latency is 12δ due to the convoy effect. In this section we present a protocol that lowers the collision-free latency to 3δ and the failure-free latency to 5δ by weaving together Skeen's protocol across groups and a Paxos-like protocol within each group. In particular, the protocol narrows the 2x gap between the collision-free and failure-free latency characteristic of existing fault-tolerant variants of Skeen's protocol. In §VI we experimentally demonstrate that these theoretical characteristics are reflected in practical performance pay-offs.

$\mathsf{clock} \leftarrow 0 \in \mathbb{N}$
$\mathsf{Phase}[\,] \leftarrow (\lambda k.\,\textsc{start}) \in$
$\quad \mathcal{M} \to \{\textsc{start}, \textsc{proposed}, \textsc{accepted}, \textsc{committed}\}$
$\mathsf{LocalTS}[\,] \in \mathcal{M} \to (\mathbb{N} \times \mathcal{G})$
$\mathsf{GlobalTS}[\,] \in \mathcal{M} \to (\mathbb{N} \times \mathcal{G})$
$\mathsf{Delivered} \leftarrow (\lambda k.\,\textsc{false}) \in \mathcal{M} \to \{\textsc{false}, \textsc{true}\}$
$\mathsf{status} \in \{\textsc{leader}, \textsc{follower}, \textsc{recovering}\}$
$\mathsf{cballot} \leftarrow \bot \in (\mathbb{N} \times \mathcal{P}) \cup \{\bot\}$
$\mathsf{ballot} \leftarrow \bot \in (\mathbb{N} \times \mathcal{P}) \cup \{\bot\}$
$\mathsf{Cur\_leader}[\,] \in \mathcal{G} \to \mathcal{P}$
$\mathsf{max\_delivered\_gts} \leftarrow \bot \in (\mathbb{N} \times \mathcal{G}) \cup \{\bot\}$

Fig. 3. Variables of a process in the white-box multicast protocol.

We list the variables maintained by our protocol in Figure 3, give its pseudocode in Figure 4, illustrate the message flow of the protocol in Figure 5 and summarise the key invariants used in its proof of correctness in Figure 6.

Preliminaries. Every process in a group is either the *leader* of the group or a *follower*. If the leader fails, one of the followers takes over. A major design decision we take in our protocol is to use the *passive replication* approach [21, 28]: *only the leader computes the timestamps and decides when to deliver an application message*. Followers are passive: they merely store the leader's decisions, so that upon the leader failure a new leader could recover the information necessary to continue multicast. A process maintains the same variables as in Skeen's protocol (Figure 1) and a few additional ones. A status variable records whether the process is a LEADER, a FOLLOWER or is in a special RECOVERING state used during leader changes. A period of time when a particular process p_i acts as a leader is denoted using a *ballot* (n, p_i)—a pair of an integer n and the process identifier p_i. Ballots are ordered lexicographically using an arbitrary total order on processes, with a special ballot $\bot$ being the minimal ballot. For a ballot

```
1  multicast(m)
2      send MULTICAST(m) to {Cur_leader[g] | g ∈ dest(m)};

3  when received MULTICAST(m)
4      pre: status = LEADER;
5      if Phase[m] = START then
6          clock ← clock + 1;
7          LocalTS[m] ← (clock, g_0);
8          Phase[m] ← PROPOSED;
9      send ACCEPT(m, g_0, cballot, LocalTS[m]) to dest(m);

10 when received ACCEPT(m, g, Bal(g), Lts(g))
       for every g ∈ dest(m)
11     pre: status ∈ {FOLLOWER, LEADER} ∧
            cballot = Bal(g_0);
       if Phase[m] ∈ {START, PROPOSED} then
12         Phase[m] ← ACCEPTED;
13     LocalTS[m] ← Lts(g_0);
14     clock ← max{time(max{Lts(g) | g ∈ dest(m)}),
                    clock};
15     forall g ∈ dest(m) do
16         send ACCEPT_ACK(m, g_0, Bal) to leader(Bal(g));

17 when received ACCEPT_ACK(m, g, Bal)
       from a quorum of p_j ∈ g in each g ∈ dest(m)
       including myself and previously received
       ACCEPT(m, g, Bal(g), Lts(g)) for every g ∈ dest(m)
18     pre: status = LEADER ∧ cballot = Bal(g_0);
19     GlobalTS[m] ← max{Lts(g) | g ∈ dest(m)};
20     Phase[m] ← COMMITTED;
21     forall {m' | Phase[m'] = COMMITTED ∧
                    Delivered[m'] = FALSE ∧
                    ∀m''. Phase[m''] ∈ {PROPOSED, ACCEPTED}
                          ⟹ LocalTS[m''] > GlobalTS[m']}
           ordered by GlobalTS[m'] do
22         Delivered[m'] ← TRUE;
23         send DELIVER(m', cballot,
                LocalTS[m'], GlobalTS[m']) to g_0;

24 when received DELIVER(m, b, lts, gts)
25     pre: status ∈ {FOLLOWER, LEADER} ∧
            cballot = b ∧ max_delivered_gts < gts;
26     Phase[m] ← COMMITTED;
27     LocalTS[m] ← lts;
28     GlobalTS[m] ← gts;
29     clock ← max{clock, time(gts)};
30     max_delivered_gts ← gts;
31     deliver(m);

32 function retry(m)
33     pre: Phase[m] ∈ {PROPOSED, ACCEPTED};
34     send MULTICAST(m) to {Cur_leader[g] | g ∈ dest(m)};

35 function recover()
36     send NEWLEADER(any ballot of the form (_, p_i)
                      higher than ballot) to g_0;

37 when received NEWLEADER(b) from p_j
38     pre: b > ballot;
39     status ← RECOVERING;
40     ballot ← b;
41     send NEWLEADER_ACK(ballot, cballot, clock,
           Phase, LocalTS, GlobalTS) to p_j;

42 when received NEWLEADER_ACK(b, cballot(p_j),
       clock(p_j), Phase(p_j), LocalTS(p_j), GlobalTS(p_j))
       from a quorum of p_j ∈ g_0
43     pre: status = RECOVERING ∧ ballot = b;
44     reinitialise Phase, LocalTS, GlobalTS;
45     var J ← the set of j with maximal cballot(p_j);
46     forall m do
47         if ∃j. Phase(p_j)[m] = COMMITTED then
48             Phase[m] ← COMMITTED;
49             LocalTS[m] ← LocalTS(p_j)[m];
50             GlobalTS[m] ← GlobalTS(p_j)[m];
51         else if ∃j ∈ J. phase(p_j)[m] = ACCEPTED then
52             Phase[m] ← ACCEPTED;
53             LocalTS[m] ← LocalTS(p_j)[m];
54     clock ← max{clock(p_j)};
55     cballot = b;
56     send NEW_STATE(b, clock, Phase, LocalTS, GlobalTS)
           to g_0 \ {p_i};

57 when received
       NEW_STATE(b, clock, Phase, LocalTS, GlobalTS)
       from p_j
58     pre: status = RECOVERING ∧ ballot = b;
59     status ← FOLLOWER;
60     cballot ← b;
61     clock ← clock; Phase ← Phase;
           LocalTS ← LocalTS; GlobalTS ← GlobalTS;
62     send NEWSTATE_ACK(b) to p_j;

63 when received NEWSTATE_ACK(b)
       from a set of processes that
       together with p_i form a quorum in g_0
64     if status = RECOVERING ∧ ballot = b then
65         status ← LEADER;
66         forall {m' | Phase[m'] = COMMITTED ∧
                        ∀m''. Phase[m''] = ACCEPTED
                              ⟹ LocalTS[m''] > GlobalTS[m']}
               ordered by GlobalTS[m'] do
67             Delivered[m'] = TRUE;
68             send DELIVER(m', cballot,
                    LocalTS[m'], GlobalTS[m']) to g_0;
```

Fig. 4. White-box multicast protocol at a process $p_i \in g_0$.

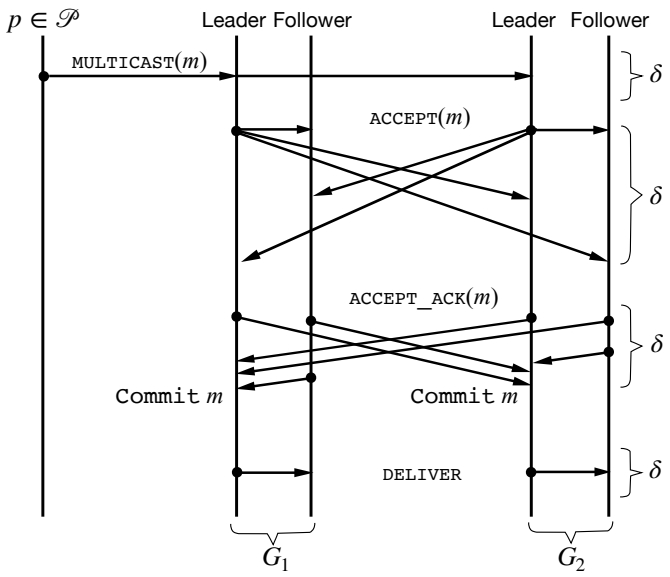

Fig. 5. Message-flow diagram illustrating the behaviour of the white-box protocol in a collision-free scenario. On the right-hand side we give the maximum time each protocol step can take.

$b = (n, p_i)$ we let $\mathsf{leader}(b) = p_i$. At any given time, a process participates in a single ballot, which is stored in a variable cballot and never decreases. During leader changes we also use an additional ballot variable ballot.

Normal operation. To multicast an application message m, a process sends it in a MULTICAST message to current leader of every group $g \in \mathsf{dest}(m)$ (line 1), which is determined using a mapping Cur_leader. This mapping need only give a guess as to the identity of the current leaders. If the guess is wrong, the multicasting process can always send the message to all the processes in a given group to find out who its leader is (omitted from the pseudocode).

A process p_i handles the message only when it is indeed the leader of its group g_0 (line 3). When the leader receives m for the first time (line 5), it performs the same actions as in Skeen's protocol (lines 9-11 in Figure 1): it increments the clock, computes the local timestamp, and sets the phase of m to PROPOSED.

Like in Skeen's protocol, the leader's next goal is to communicate its local timestamp proposal to the leaders of the other destination groups of m, so that all leaders could compute the global timestamp and deliver the message. A key idea used to achieve fault-tolerance and reduced latency in our protocol is not to send local timestamps to the leaders directly, but *route them through a quorum of processes in each destination group, to ensure their durability.* Namely, the leader sends an ACCEPT message including its ballot and the computed local timestamp to all processes in $\mathsf{dest}(m)$ (including itself, for uniformity, line 9); this message is analogous to the "2a" message of Paxos. As we explain in the following, due to failures the leader may receive the same MULTICAST(m) message twice. In this case the leader just resends the ACCEPT message with the locally stored data for m. This ensures Invariant 1: in a given ballot, a message can be assigned at most one local timestamp.

A process that is a destination of m acts once it receives an ACCEPT message for m from the leader of each of the destination groups $g \in \mathsf{dest}(m)$ (line 10). The message carries the local timestamp proposal $Lts(g)$ and the ballot $Bal(g)$ of the leader making the proposal. The process checks that it participates in the ballot $Bal(g_0)$ of the leader of its group g_0 it received the message from. Then the process advances the phase of the message m to ACCEPTED, stores its local timestamp in the LocalTS array (line 13) and ensures its clock is no lower than the global timestamp obtained by taking the maximum of the local timestamps $Lts(g)$ of m (line 14). Lines 13 and 14 in our protocol can be viewed as replicating lines 10 and 15 of Skeen's protocol (Figure 1) throughout the process group. The process acknowledges the acceptance of the local timestamps by sending an ACCEPT_ACK message to the leaders who made the proposals, tagged with the vector of ballots Bal in which these proposals were made at the destination groups; this message is analogous to the "2b" message of Paxos.

A leader who made a local timestamp proposal for m waits until it receives a quorum of ACCEPT_ACK messages for m with matching ballot vectors from each of the destination groups $\mathsf{dest}(m)$ (line 17); Invariant 1 ensures that the different ACCEPT_ACK messages correspond to the same set of local timestamp proposals. At this point the leader considers that all local timestamps for m are agreed, and thus it advances the phase of m to COMMITTED, computes its final global timestamp as the maximum of the local timestamps and stores it in the GlobalTS array. The leader then tries to deliver one or more committed messages like in Skeen's protocol, in the order of their global timestamps (line 21, corresponding to line 17 in Figure 1). To this end, it sends the data about each message m' to deliver in a DELIVER message to all the members of its group.

Since our communication channels are FIFO, during failure-free execution a process receives DELIVER messages in the order the leader of its group sends them. Upon receiving such a message, the process stores the enclosed information and delivers the corresponding application message. As we explain in the following, when failures occur, a process may receive duplicate DELIVER messages. To handle this, each process maintains the highest global timestamp of an application message it has delivered in a variable max_delivered_gts and ignores DELIVER messages carrying lower global timestamps.

Discussion of normal operation. As we mentioned earlier, our optimised protocol can be viewed as weaving together the steps from Skeen's protocol and Paxos. In particular, when multicasting a local application message m with $\mathsf{dest}(m) = \{g_0\}$, the protocol exactly follows the flow of Paxos: the leader of g_0 sends a proposal to all processes in g_0 (ACCEPT) and waits for a quorum of acknowledgements (ACCEPT_ACK), whereupon it delivers m (DELIVER). Like in Paxos, when a process receives the ACCEPT message from the leader (line 10),

the process checks that it participates in the ballot the leader is in (line 11), thus ensuring that it only stores local timestamps (line 13) issued by the leader it supports.

For a global application message, the flow of the protocol is also similar to the one of Paxos, but performed between multiple leaders on the one hand and multiple groups of followers on the other. However, note that a process does not perform any checks on ballots in ACCEPT messages received from remote groups (line 10); these ballots are only used in line 17 to ensure that different ACCEPT_ACK messages correspond to the same set of local timestamp proposals. Hence, the ACCEPT messages may well come from old leaders of remote groups that have since been deposed and whose local timestamp proposals will be rejected by their groups. The update to the clock at line 14 may thus be performed based on such invalid local timestamps. A key insight used in our protocol is that this situation does not violate correctness. The clock variables at processes of the same group are used to simulate the clock variable of a reliable process in Skeen's protocol: as we explain in the following, if the group leader fails, a new leader recovers the clock value from the clocks at followers. But as we noted in §III, *the* clock *variable in Skeen's protocol can always be safely increased.*

Hence, the Paxos-like ACCEPT and ACCEPT_ACK messages in our protocol can be viewed as replicating in one go both the local timestamp assignment (line 10 in Figure 1) and the clock increase (line 15 in Figure 1), with the latter done speculatively, before the local timestamps are agreed. Once a leader receives a quorum of ACCEPT_ACK messages from each of the destination groups (line 17 of our protocol), it knows that the clocks at the processes in these quorums have already been advanced to be no lower than the corresponding global timestamp. The leader can thus avoid a round trip to replicate the clock update, required in the naive fault-tolerant version of Skeen's protocol we presented earlier. The leader then replicates the global timestamps off the critical path, in DELIVER messages, by exploiting the fact that global timestamps are uniquely determined by local timestamps.

As we argue in §V, the normal processing in the protocol has the collision-free latency of 3δ, which reflects the length of the communication path from the process multicasting a message to its delivery at a leader (MULTICAST, ACCEPT, ACCEPT_ACK); in contrast, the failure-free latency is 5δ. Since followers deliver an application message only after receiving a DELIVER message from their leader, the maximum time to deliver a message is bounded by 4δ in a collision-free run and 5δ in a failure-free one.

Key invariants. We now describe the key invariants of the protocol used to prove its correctness, which also motivate the design of recovery from leader failures. Invariant 2 ensures that, if a quorum of processes in a group g_0 accepted the same set of local timestamp proposals Lts for an application message m, then the message m and its local timestamp $Lts(g_0)$ at g_0 will persist in all ballots higher than the ballot $Bal(g_0)$ at which g_0 accepted them (a, b); furthermore, the clock values at these ballots will be no lower than the global timestamp computed from the local timestamp proposals Lts for m (c). Lines 12, 13 and 14 in our protocol contribute to preserving the clauses (a), (b) and (c) of the invariant, respectively. Since Invariant 2(a, b) ensures that local timestamps accepted by a quorum persist across leader changes, we then get Invariant 3(a), ensuring that each group agrees on the local timestamp of a given application message. Since the global timestamp for an application message is computed as the maximum of local timestamps accepted by quorums in each destination group, from Invariant 3(a) we get Invariant 3(b), ensuring that the system agrees on the global timestamp of each message. Finally, similarly to how it was done for Skeen's protocol, we can show Invariant 4, ensuring that global timestamps are unique for each message.

1) For any two messages sent of the form $\texttt{ACCEPT}(m, g, b, lts_1)$ and $\texttt{ACCEPT}(m, g, b, lts_2)$, we must have $lts_1 = lts_2$.
2) Assume that at some point a quorum of processes in g_0 have received the set of messages

$$\{\texttt{ACCEPT}(m, g, Bal(g), Lts(g)) \mid g \in \mathsf{dest}(m)\} \quad (1)$$

and responded to them with

$$\texttt{ACCEPT\_ACK}(m, g_0, Bal). \quad (2)$$

Whenever at a process in g_0 we have $\mathsf{cballot} > Bal(g_0)$, we also have:
   a) $\mathsf{Phase}[m] \in \{\text{ACCEPTED}, \text{COMMITTED}\}$;
   b) $\mathsf{LocalTS}[m] = Lts(g_0)$;
   c) $\mathsf{clock} \geq \mathsf{time}(\max\{Lts(g) \mid g \in \mathsf{dest}(m)\})$;
3) a) For any messages $\texttt{DELIVER}(m, \_, lts_1, \_)$ and $\texttt{DELIVER}(m, \_, lts_2, \_)$ sent to processes in the same group, we have $lts_1 = lts_2$.
   b) For any messages $\texttt{DELIVER}(m, \_, \_, gts_1)$ and $\texttt{DELIVER}(m, \_, \_, gts_2)$ sent to any groups, we have $gts_1 = gts_2$.
4) For any $\texttt{DELIVER}(m_1, \_, \_, gts_1)$ and $\texttt{DELIVER}(m_2, \_, \_, gts_2)$ messages sent, if $m_1 \neq m_2$, then $gts_1 \neq gts_2$.
5) Assume that at some point a quorum of processes in g_0 have received the set of messages (1) and responded to them with (2) and that this quorum includes $\mathsf{leader}(Bal(g_0))$. Let $gts = \max\{Lts(g) \mid g \in \mathsf{dest}(m)\}$ and let $LocalTS_0$ be the projection of $\mathsf{LocalTS}$ when $\mathsf{leader}(Bal(g_0))$ sent its ACCEPT_ACK to messages m' such that $\mathsf{Phase}[m'] \neq \text{START} \wedge \mathsf{LocalTS}[m'] < gts$. Whenever at a process in g_0 we have $\mathsf{cballot} > Bal(g_0)$, we also have:

$$\forall m'.\, \mathsf{Phase}[m'] \neq \text{START} \wedge \mathsf{LocalTS}[m'] < gts \implies \mathsf{LocalTS}[m'] = LocalTS_0[m']. \quad (3)$$

Fig. 6. Key invariants of the white-box multicast protocol.

Finally, Invariant 5 ensures that application messages are delivered in the order of their global timestamps, despite leader changes. Similarly to Invariant 2, this invariant assumes that a quorum of processes in a group $g_0 \in \mathsf{dest}(m)$, including its leader $\mathsf{leader}(Bal(g_0))$, have accepted the same set of local timestamp proposals Lts for m, yielding a global timestamp gts. The invariant ensures that, in any future ballot of group g_0, a process may not have messages with local timestamps less than gts that the leader $\mathsf{leader}(Bal(g_0))$ did not know about when it accepted the local timestamp for m. Given this invariant and the check on local timestamps the leader performs before delivering an application message (line 21), if a leader of a group g_0 delivers a message m with a global timestamp gts, then it can be sure that no message it is not aware of will get a local timestamp lower than gts in future ballots, and thus no message will get a lower global timestamp.

Invariant 5 is proved using Invariant 2(c): under the assumptions of the former invariant, the latter one ensures that the clock of any leader of a future ballot will be no lower than gts. Then any new application message this leader receives will get a local timestamp at g_0 higher than gts.

Leader recovery. The leader of each group $g \in \mathcal{G}$ is continuously monitored by its followers, which trigger leader election whenever the current leader is suspected as faulty. The leader monitoring and election protocol exploits the knowledge of the upper bound on the failure-free message propagation delay δ to guarantee that there exists a time $t \geq \mathsf{GST}$ after which the same correct process is permanently trusted as g's leader by all members of g. Examples of leader election protocols satisfying this property can be found in [5, 7, 25, 26].

The leader recovery procedure is activated whenever the existing leader is replaced with a new one by the leader election protocol. Its main goal is to preserve Invariants 2 and 5. Ensuring the latter is particularly subtle: for this, *before the new leader starts multicast, it must bring a quorum of followers in sync with its state* (this is similar to [21, 28]). Hence, a new leader is elected in two stages. First, processes vote to join the ballot of a prospective leader, which they record in a variable ballot; like cballot, this variable can only increase. Second, processes receive and acknowledge an initial state from the new leader and set cballot to $ballot$. The leader only resumes normal operation after it gets a quorum of such acknowledgements. Note that we thus always have $\mathsf{cballot} \leq \mathsf{ballot}$.

In more detail, when a process p_i is elected a leader, it invokes the `recover` function (line 35), which attempts to establish a new ballot with p_i as its leader. The process picks a ballot that it leads such that it is higher than the last ballot it joined and sends it in a `NEWLEADER` message to the group members (including itself); this message asks the group members to support the process as the new leader and is analogous to the "1a" message in Paxos. When a process receives a `NEWLEADER`(b) message (line 37), it first checks that the proposed ballot b is higher than the last ballot it joined. In this case it sets ballot to b and changes its status to RECOVERING, which causes it to stop normal message processing (due to the guards at lines 11, 18 and 25). The process then replies to the new leader with a `NEWLEADER_ACK` message containing all components of its state; this message serves as a vote for the new leader and is analogous to the "1b" message of Paxos.

The new leader waits until it receives `NEWLEADER_ACK` messages from a quorum of group members (line 42). Based on the states reported in them, it computes a new state from which to resume multicast according to the following rules. First, if an application message m is COMMITTED at some process, then the leader marks it as COMMITTED and copies its local and global timestamps (line 47). If a message m is not COMMITTED at any process, then, like in Paxos, the leader looks at the states of processes that reported the maximal cballot (line 45): if a message m is ACCEPTED at such a process, then the leader marks it as ACCEPTED and copies its local timestamp (line 51). Like for Paxos, we can show that these rules preserve Invariant 2(a, b). Finally, the leader sets clock to the maximum of the clock values reported by processes, to preserve Invariant 2(c), and sets cballot to the new ballot.

The new leader next ensures that at least a quorum of processes in its group are in sync with its new state. To this end, it sends a `NEW_STATE` message with the new state to the other group members (line 56). Upon receiving this message (line 57), a process overwrites its state with the one provided, changes its status to FOLLOWER, and sets cballot to b, thereby recording the fact that it has synchronised with the leader of b. The process then replies to the new leader with a message `NEWSTATE_ACK`(b) confirming this.

The new leader waits until it receives `NEWSTATE_ACK` from a set of processes that together with it form a quorum (line 63). The leader may have application messages ready to be delivered that some of the followers have not delivered yet. In fact, different followers may have delivered different sequences of application messages, because the previous leader may have crashed in between sending `DELIVER` messages to different followers. To deal with this, the leader delivers all the committed messages it can, starting from the beginning. This does not violate correctness since, as we explained earlier, followers check for duplicate `DELIVER` messages using the max_delivered_gts variable. At the end, the new leader sets status to LEADER, which allows it to start normal operation.

Discussion of leader recovery. We now highlight some of the subtleties of the recovery procedure. First, note that upon a leader change, the value of the clock at leaders may actually decrease. For example, assume a process $p_i \in g_0$ is a leader who issued a local timestamp (t, g_0) for an application message m and thus set $\mathsf{clock} = t$. If p_i fails before a quorum of processes in g_0 accepts m, the new leader may derive its initial state from a quorum of processes that did not see m and end up with a clock value lower than t. This does not violate correctness: to ensure that messages are delivered in the order of their timestamps we only need to ensure that the

clock does not fall below the global timestamp of a message accepted by a quorum, as stated by Invariant 2(c).

We next illustrate why it is important for a leader to synchronise its state with the followers before starting normal operation. Assume a process $p_1 \in g_0$ is a leader of a ballot b_1 who has issued a local timestamp lts for an application message m and replicated it to some of its followers in g_0. Assume further that before p_1 manages to reach a quorum, another process $p_2 \in g_0$ becomes the leader at a ballot $b_2 > b_1$. To compute its initial state, the process p_2 may query a quorum that does not contain any processes that saw m and lts, so that its initial state will exclude these. Assume that at a later point p_2 commits and delivers a message m' with a global timestamp $gts' > lts$. Now imagine there is yet another leader change and a process p_3 becomes a leader at a ballot $b_3 > b_2$. Since before p_2 delivered m', it got a quorum of followers to accept its initial state and set $\mathsf{cballot} = b_2$, when p_3 queries a quorum to compute its initial state, it is guaranteed to see at least one process with $\mathsf{cballot} = b_2$; this process will report a state excluding m and lts. According to the rule used to compute the initial state in line 51, p_3 will then disregard any processes that accepted m and lts at ballot $b_1 < b_2$. This will ensure Invariant 5: the local timestamp lts for m, which the leader p_2 did not know about when it committed m', will never be resurrected upon recovery. Hence, the message m will never be able to get a timestamp lower than gts', and the decision by p_2 to deliver m' will stay valid.

Message recovery. In the above scenario message m gets lost at the group g_0 due to a leader failure. Even if other destination groups have received it, its processing will not progress. To deal with this situation, the multicasting process can just resend the $\texttt{MULTICAST}(m)$ message. Then groups that have not previously received m will start processing it, and groups that have already processed m will just resend the corresponding protocol messages (lines 9 and 16), which will unblock the processing of m.

The processing of a message m can also get stuck if the process submitting it for multicast fails in between sending $\texttt{MULTICAST}(m)$ messages to different leaders (line 2), so that one group $g_1 \in \mathsf{dest}(m)$ receives m and another group $g_2 \in \mathsf{dest}(m)$ does not receive it. This will cause m to get stuck in the PROPOSED phase at the leader of g_1, since the group g_2 will never send a local timestamp proposal for m. The leader of g_1 can again recover from this situation by resending the $\texttt{MULTICAST}(m)$ message to all destination groups of m (line 34). The same mechanism can be used to resume the processing of an accepted message after a leader change.

V. Correctness and Latency Analysis

In [18] we prove

Theorem 2: The white-box protocol in Figure 4 is a correct and genuine implementation of atomic multicast.

We prove the Ordering, Validity and Integrity properties using the invariants in Figure 6. To prove Termination, we rely on the property of the leader election and monitoring protocol running in every group $g \in \mathcal{G}$, which guarantees that after GST, all correct processes permanently trust the same correct member of g as their leader. In particular, this allows us to prove the following key lemma:

Lemma 1: There exists a time $t \geq \mathsf{GST}$ such that starting from t onward, every application message m received by the leader p_i of some $g \in \mathsf{dest}(m)$ is either already committed at p_i, or will commit at p_i within the time 3δ since it was multicast.

From the above we conclude

Theorem 3: The collision-free latency of the white-box protocol in Figure 4 is 3δ.

The failure-free latency (*FFL*) of all atomic multicast protocols that fit the general framework of Skeen's protocol in Figure 1 is given by

$$FFL = CFL + C, \tag{4}$$

where *CFL* is the protocol's collision-free latency, and C is the upper bound on the time elapsing between $\texttt{multicast}(m)$ and the time at which a process in $\mathsf{dest}(m)$ advances its clock past $\mathsf{GlobalTS}[m]$ in a failure-free run.

To see why, consider time $t \geq \mathsf{GST}$ stipulated by Lemma 1. Let m be an application message multicast at $t_1 > t$, and p_i be the leader of some group $g \in \mathsf{dest}(m)$ that delivers m at $t_2 > t_1$. By the protocol, both of the following conditions must hold at t_2: *(i)* m is committed at p_i, and *(ii)* all messages m' such that the $\mathsf{LocalTS}[m'] < \mathsf{GlobalTS}[m]$ are committed at p_i. Let t' be the time at which p_i advances its clock past $\mathsf{GlobalTS}[m]$. From *(ii)*, we have $t_1 \leq t' \leq t_2$. Note that any application messages received by p_i after t' will have its local timestamps $> \mathsf{GlobalTS}[m]$ at p_i, and therefore, will not be blocking the m's delivery. Since by Lemma 1, all messages received by p_i before t' are either already committed or will commit within 3δ, we conclude that $t_2 \leq t' + 3\delta$. Since $C = t' - t_1$, we have $t_2 - t_1 \leq CFL + C$. Given that for the white-box protocol, $C = 2\delta$, we receive

Theorem 4: The failure-free latency of the white-box protocol in Figure 4 is 5δ.

VI. Experimental Evaluation

We have implemented our multicast protocol in C using the `libevent` library for communication [1]. Our implementation is available at [2]. In addition to the protocol described in §IV, the implementation includes a mechanism to garbage collect delivered messages. In this section we experimentally compare our protocol with the naive fault-tolerant version of Skeen's we described in §IV [17] and a state-of-the-art *FastCast* protocol by Coelho et al. [10]. We use open-source implementations of these protocols by Coelho et al. [3], also implemented in C and using `libevent`.

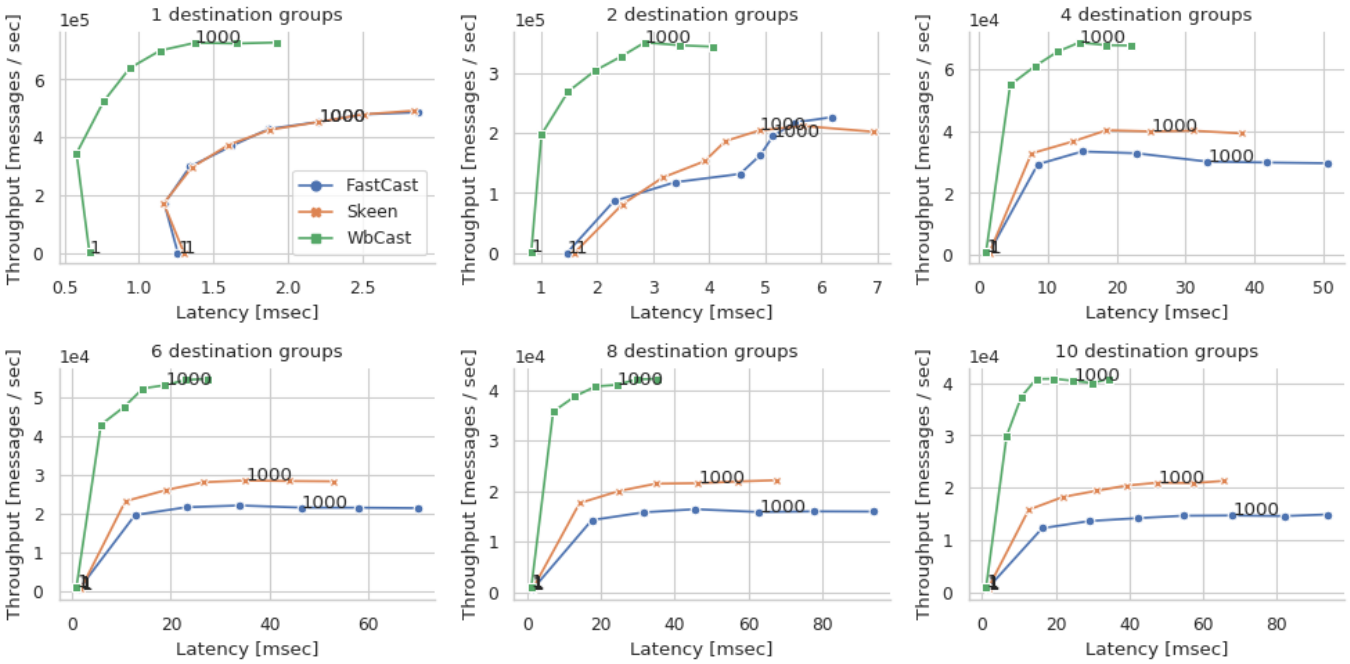

Fig. 7. Performance of multicast protocols in LAN with increasing numbers of clients: FastCast, fault-tolerant Skeen and our protocol (WbCast). In each experiment clients multicast messages to a fixed number of groups. For reference, we mark the points corresponding to 1000 clients.

Competitor protocols. Both of fault-tolerant Skeen's protocol and FastCast use consensus as a black box; the former protocol has collision-free latency of 6δ, and the latter of 4δ. FastCast optimises fault-tolerant Skeen's protocol by using speculative execution. Like in Skeen's, upon receiving an application message, the Paxos leader of a group issues a tentative local timestamp based on its local clock and invokes consensus to persist it. However, the leader also immediately sends the local timestamp to the leaders of the other destination groups, without waiting for consensus to finish. The leaders then speculatively act on these timestamps like in Skeen's, computing the global timestamp as their maximum, advancing their clocks in line with it and invoking consensus to persist these actions. Once the consensus on the local timestamps is reached, the leaders exchange messages confirming this. By the time a leader receives these messages, it may have already done all of the work necessary to act on the local timestamps, and can commit the corresponding application message at once. In the absence of failures (or suspicions thereof) the speculative execution always succeeds, resulting in collision-free latency of 4δ.

In comparison to this protocol, ours avoids using separate consensus calls to replicate a local timestamp and to advance the clock above a global timestamp, resulting in collision-free latency of 3δ. Furthermore, in both fault-tolerant Skeen's protocol and FastCast, the clock is advanced past the message's global timestamp only after the second consensus has been completed, which, by (4), implies that their failure-free latencies are 2x their respective collision-free latencies, i.e., 8δ and 12δ respectively. In contrast, our speculative clock update mechanism allows the leader to advance its clock to the future global timestamp of a message immediately upon receiving a full set of local timestamp proposals from all its destination groups thus achieving the failure-free latency of 5δ.

Local-area network. We first benchmark the protocols in a local-area network (LAN) using the CloudLab infrastructure [4]. We consider 10 groups, each with 3 replicas, residing on 30 machines. A varying numbers of client processes residing on 10 separate machines initiate multicasts of 20-byte messages. We use machines with Xeon E5-2640 10-core processors and 64GB of memory, connected with 2GB network links with around 0.1ms round-trip time.

We follow the evaluation methodology similar to the one previously used to benchmark FastCast [10]. In Figure 7 we show the average latency and throughput as a function of the number of clients and the number of destination groups these clients multicast to. We give the case where clients multicast to all groups only for completeness: this is not an intended deployment of genuine atomic multicast, since atomic broadcast performs better in this situation [33].

As is evident from Figure 7, our protocol consistently outperforms FastCast and Skeen both in latency and in throughput. For example, at 1000 clients with respect to FastCast our protocol improves both latency and throughput by 70-150%, depending on the number of destination groups. Note that in LAN, FastCast generally performs slightly worse than Skeen. This is consistent with the results in [10] and is due to the overhead of introduced by its parallel execution paths: this protocol is more suited for a wide-area network.

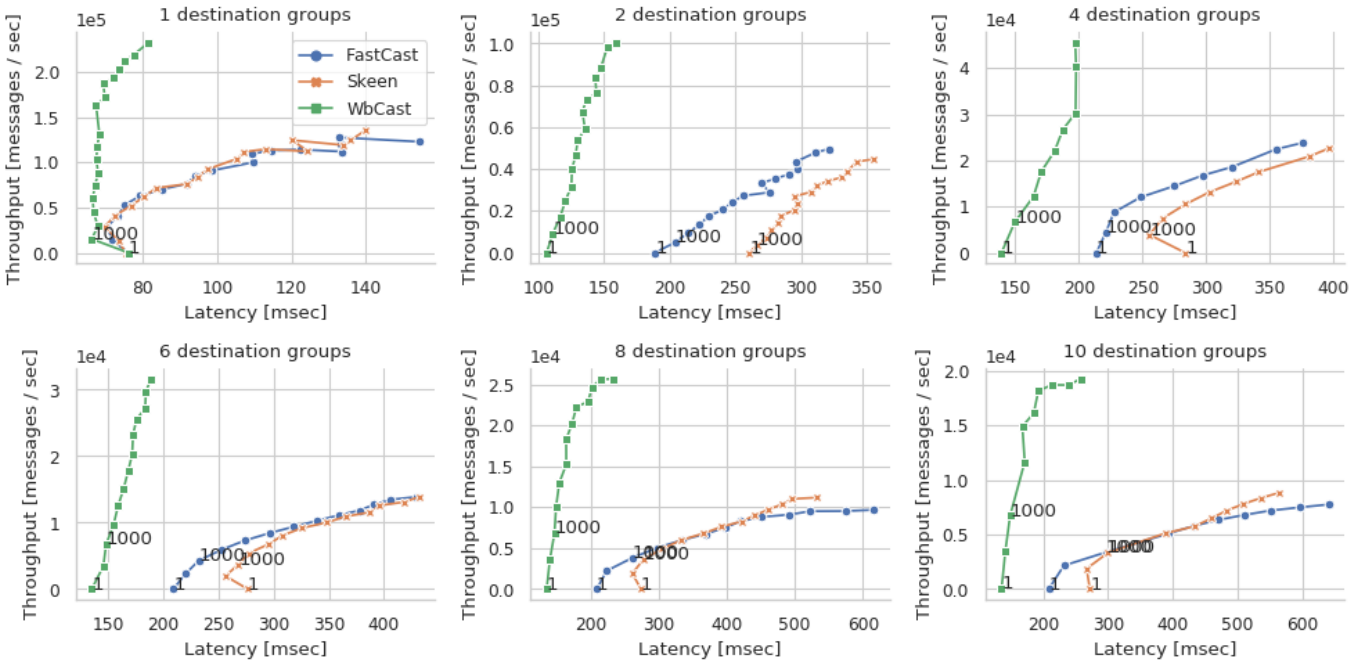

Fig. 8. Performance of multicast protocols in WAN with increasing numbers of clients: FastCast, fault-tolerant Skeen and our protocol (WbCast). In each experiment clients multicast messages to a fixed number of groups. For reference, we mark the points corresponding to 1000 clients.

Wide-area network. We next benchmark the protocols in a wide-area network (WAN). We again consider 10 groups replicated across 3 data centres on the Google Cloud Platform. Each group has a replica in each data centre, so that a single data centre contains a complete copy of the data managed by the system. This setting is typical for modern wide-area deployments [11]. The data centres are Oregon (R1), North Virginia (R2) and England (R3), and average round-trip times between them are 60ms (R1↔R2), 75ms (R2↔R3) and 130ms (R1↔R3). We use machines with 2 vCPUs and 7.5GB of memory for multicast group members, and 8 vCPUs and 30GB of memory to generate client load.

In Figure 7 we show the performance of all protocols in this environment. Our protocol again outperforms both FastCast and Skeen. For example, at 1000 clients it outperforms FastCast on both latency and throughput by 47%-124%, depending on the number of destination groups. It can also sustain higher throughput at higher numbers of clients: by 140% for 6 destination groups.

VII. Related Work

Genuine atomic multicast is often implemented using a fault-tolerant version of Skeen's protocol [17, 31], which has the collision-free latency of 6δ. Early alternatives had asymptotically worse time complexity, e.g., proportional to the number of destination groups [14]. As this is unsatisfactory, researchers have been looking for protocols with lower latency. Rodrigues et al. [29] proposed a protocol that has the collision-free latency of 5δ. More recently, Coelho et al. [10] proposed the FastCast protocol that further lowers it to 4δ, which we discussed in detail in §VI. Our protocol has the collision-free latency of 3δ. It also boasts a lower failure-free latency of just 5δ thus reducing the 2x latency degradation caused by concurrent messages in all previously proposed implementations of atomic multicast based on Skeen's protocol.

Our experimental results demonstrate that minimising latency is not only of theoretical interest, but enables superior performance in practice. The above protocols also used consensus as a black-box, whereas take a different approach, unpacking Paxos and weaving it together with Skeen's protocol.

In this paper we assumed that each group has enough correct processes to function normally. Researchers have also investigated atomic multicast protocols that can operate when a whole group crashes [32]. We also assumed that process failures are crash-stop, rather than Byzantine [9]. We leave handling these more challenging cases for future work.

Another primitive whose fault-tolerance presents similar challenges to atomic multicast is *atomic commit*, which allows several process groups to reach a decision on whether a database transaction should be committed or aborted. A naive fault-tolerant solution to this problem layers the classical two-phase commit protocol over Paxos [11]. There have been several alternative proposals that reduce the latency by developing a single coherent protocol, in the spirit of this work [8, 22, 36]. In comparison to these proposals, we handle the more challenging problem of atomic multicast, where process groups need to agree on a total ordering of application messages rather than on a binary per-transaction decision. This required us to develop new techniques for replicating operations on logical clocks in a latency-conscious way.

ACKNOWLEDGEMENTS

We thank Manuel Bravo, Thanh Hai Tran and Pierre Sutra for helpful comments and discussions. We also thank Paulo Coelho and Fernando Pedone for discussions about their FastCast protocol. Alexey Gotsman was supported by an ERC grant RACCOON.

REFERENCES

[1] https://libevent.org/.
[2] https://github.com/imdea-software/?????
[3] https://bitbucket.org/paulo_coelho/libmcast.
[4] https://www.cloudlab.us/.
[5] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *International Conference on Distributed Computing (DISC)*, 2001.
[6] T. Ahmed-Nacer, P. Sutra, and D. Conan. The convoy effect in atomic multicast. In *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*, pages 67–72, Sep. 2016.
[7] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.
[8] G. Chockler and A. Gotsman. Multi-shot distributed transaction commit. In *International Symposium on Distributed Computing (DISC)*, 2018.
[9] P. R. Coelho, T. C. Junior, A. Bessani, F. L. Dotti, and F. Pedone. Byzantine fault-tolerant atomic multicast. In *International Conference on Dependable Systems and Networks (DSN)*, 2018.
[10] P. R. Coelho, N. Schiper, and F. Pedone. Fast atomic multicast. In *International Conference on Dependable Systems and Networks (DSN)*, 2017.
[11] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.
[12] J. A. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference (USENIX ATC)*, 2012.
[13] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.
[14] C. Delporte-Gallet and H. Fauconnier. Fault-tolerant genuine atomic multicast to multiple groups. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2000.
[15] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
[16] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
[17] U. Fritzke Jr., P. Ingels, A. Mostéfaoui, and M. Raynal. Consensus-based fault-tolerant total order multicast. *IEEE Trans. Parallel Distrib. Syst.*, 12(2), 2001.
[18] A. Gotsman, A. Lefort, and G. Chockler. White-box atomic multicast (extended version). *arXiv CoRR*, ?????, 2019. Available from http://arxiv.org/abs/?????
[19] R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.*, 254(1-2), 2001.
[20] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.
[21] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *International Conference on Dependable Systems and Networks (DSN)*, 2011.
[22] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *European Conference on Computer Systems (EuroSys)*, 2013.
[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.
[24] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.
[25] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *Symposium on Reliable Distributed Systems (SRDS)*, 2000.
[26] M. Larrea, A. Fernandez, and S. Arevalo. On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Trans. Comput.*, 53(7):815–828, 2004.
[27] J. Lockerman, J. M. Faleiro, J. Kim, S. Sankaran, D. J. Abadi, J. Aspnes, S. Sen, and M. Balakrishnan. The FuzzyLog: A partially ordered shared log. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
[28] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*, 1988.
[29] L. E. T. Rodrigues, R. Guerraoui, and A. Schiper. Scalable atomic multicast. In *International Conference On Computer Communications and Networks (ICCCN)*, 1998.
[30] M. Saeida Ardekani, P. Sutra, and M. Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *Symposium on Reliable Distributed Systems (SRDS)*, 2013.
[31] N. Schiper and F. Pedone. On the inherent cost of atomic broadcast and multicast in wide area networks. In *International Conference on Distributed Computing and Networking (ICDCN)*, 2008.
[32] N. Schiper and F. Pedone. Solving atomic multicast when groups crash. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2008.
[33] N. Schiper, P. Sutra, and F. Pedone. Genuine versus non-genuine atomic multicast protocols for wide area networks: An empirical study. In *Symposium on Reliable Distributed Systems (SRDS)*, 2009.
[34] N. Schiper, P. Sutra, and F. Pedone. P-Store: Genuine partial replication in wide area networks. In *Symposium on Reliable Distributed Systems (SRDS)*, 2010.
[35] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), 1990.
[36] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Symposium on Operating Systems Principles (SOSP)*, 2015.